6-1996

An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams

Gary T. Leavens
Iowa State University

Tim Wahls
Iowa State University

Albert L. Baker
Iowa State University

Kari Lyle
Iowa State University

An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams

Abstract
Using operational semantic techniques, an extended variant of structured analysis style data flow diagrams is given a formal semantics. This semantics allows one to describe both how information is processed and the dynamic behavior of the system. The ability to describe dynamic behavior is an extension to the traditional notion of data flow diagrams. This semantics can serve as a target for giving meaning to specification languages that use a graphical notation similar to data flow diagrams.

Keywords
structured analysis, data flow diagram, operational semantics, formal specification, firing rule, store

Disciplines
Systems Architecture

An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams

Gary T. Leavens, Tim Wahls,
Albert L. Baker, and Kari Lyle

TR #93-28c
December 1993 (revised Dec. 1993, Sept. 1994, June 1996)

Keywords: structured analysis, data flow diagram, operational semantics, formal specification, firing rule, store.

1993 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodologies, tools; D.2.2 [*Software Engineering*] Tools and Techniques — structured analysis, data flow diagrams, D.2.10 [*Software Engineering*] Design — methodologies, representation; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics, syntax; F.3.2 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — specification techniques; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — operational semantics.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

An Operational Semantics of Firing Rules for Structured Analysis Style Data Flow Diagrams

Gary T. Leavens*, Tim Wahls, Albert L. Baker, and Kari Lyle
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu, wahls@howard.hbg.psu.edu, baker@cs.iastate.edu

June 6, 1996

Abstract

Using operational semantic techniques, an extended variant of structured analysis style data flow diagrams is given a formal semantics. This semantics allows one to describe both how information is processed and the dynamic behavior of the system. The ability to describe dynamic behavior is an extension to the traditional notion of data flow diagrams. This semantics can serve as a target for giving meaning to specification languages that use a graphical notation similar to data flow diagrams.

1 Introduction

An approach to the development of software systems which has enjoyed wide-spread use in the software engineering community is *Structured Analysis* (SA) [DeM78] [GS78] [WM85] [You89]. Within SA one specifies a data model using an *Entity-Relationship Diagram* (ERD) and a data dictionary, and the process by a *Data Flow Diagram* (DFD) [Fra93]. Because DFDs are widely-used [BB93], many tools support their development. There are at least three attributes of DFDs that are appealing to software engineers:

- they have a graphical representation,
- they are hierarchical, thereby supporting the kind of modular decomposition that programmers view as essential, and
- they are informal.

Since DFDs do not have a precise semantics, a DFD (even when combined with a ERD) cannot serve as a *formal* specification of the functionality of a software system. In addition, DFDs are not even intended to capture the dynamic behavior of a software process.

Various researchers have proposed ways to use SA techniques either as a first step towards extracting a formal specification, or by augmenting SA techniques with the goal

*Leavens's work is supported in part by the National Science Foundation under grants CCR-9108654 and CCR-9593168. Wahls's work was supported in part by a research assistantship provided by the College of Liberal Arts and Sciences, Iowa State University. Baker's work was supported in part by a grant from Rockwell International.

of making them more precise. For example, Fraser, Kumar, and Vaishnavi [FKV91] and Larsen, *et al.* [LvKP+91] extract a VDM [BJ82] [Jon90] specification from a DFD and a data dictionary. As another example, Semmens and Allen [SA91] extract a Z [Hay93] [Spi92] [Spi89] specification from an application's ERD and DFD. See [FLP94] for a survey that includes other such examples.

For this paper, the more relevant techniques are those that use formal notations to supplement SA techniques. For example, Wing and Zaremski [WZ91] augment SA specifications (especially the data dictionary), with specifications in the Larch Shared Language [GHG+93]. As another example, France [Fra93] [FLP95], has specified the types in a data dictionary using Z notation, and has used Z to augment the specifications of data stores and global state invariants. These extensions are relevant, because they indicate that there is a desire for integration of DFDs and formal methods.

These supplements to SA techniques motivate the problem we address in this paper: how to formally model both the standard concepts of DFDs and integrate that with a model of their dynamic behavior. Doing this would provide a semantic foundation for:

- defining extended DFD specification languages that would allow for formal specification of both data transformations and dynamic behavior, and the consequent ability to do formal reasoning and validation,
- comparing the semantics of various extended DFD specification languages, and
- using DFDs informally in ways that are more expressive and precise.

An additional benefit of our efforts is that, by focusing on the semantics of an extended variant of DFDs, we explore much of the space of possible specification languages that use DFD notations and specify dynamic behavior; such an exploration would not be possible if we were presenting a particular specification language.

Some may object to this effort on the grounds that its intention to support extended DFD specification languages is misguided. They would say that a large part of the appeal of DFDs is their informality, which allows them to be used during early stages of requirements analysis and specification. But having a formal DFD specification language will certainly not prevent anyone from drawing pictures on paper; what it will do is allow the possibility of taking such paper sketches and formalizing them without a complete change of notation. We also believe that an understanding of our semantics will aid both the design of extended DFD specification languages and aid the work of systems analysts who wish to be more precise in their use of DFDs.

Due to space considerations, we will not give the details of an example extended DFD specification language (some preliminary ideas are, however, found in [WBL93]), or a translation from such a language into our semantic model. What we offer is a definition of the target of such a translation (the semantic model), and examples of translations which the reader can use to judge the suitability of our model. Thus, in this paper we limit ourselves to explaining the formal model, a variant of DFDs, and its formal semantics.

Because we are giving a semantic foundation to extended DFD specification languages, we do not limit ourselves to exactly the traditional notations and concepts used in DFDs. (Such notations and concepts are fraught with ambiguity in any case.) Instead, we follow the example of workers in semantics of programming languages, who extract from real programming languages a simplified set of core concepts, into which a real programming language could be translated. No one expects a theoretical core language (like the λ-calculus) to be of practical value in real programming. Instead, a standard way to judge

such theoretical core languages is to see if they can capture (by translation) the meaning of all the constructs of relevant real languages. Thus, our intention is to define a model that can act as a theoretical core for DFD-like specification languages. The model should be judged by whether it can serve as a translation target for traditional DFDs and extended DFD-like specification languages. It should not be judged by how exactly it matches the traditional DFD notation.

The main extension to the traditional notions of DFD specification is the specification of dynamic behavior; that is, our model gives a formal description of a DFD's firing rules. We believe that with such an extension, DFDs may be useful for the specification of concurrent and distributed systems. In such applications, the precise dynamic behavior of the system is more important than when a DFD is used to help design a single-processor program.

In overview, the meaning our model would assign to an extended DFD specification is a set of sequences of configurations of the DFD. A configuration of a DFD tells the state of each process and flow. A sequence of configurations represents a possible execution of the DFD. The model uses sets of such sequences to handle concurrency and nondeterminism that may be allowed by the specification.

In addition to the modeling of dynamic behavior, we believe that the following aspects of our semantics are interesting in the sense of finding a smaller theoretical core for DFDs.

1. DFD terminators (external entities) have a specified behavior, which is unusual (although found in [Fra93]); however, if no constraints on the terminators are desired, then the specification can simply permit arbitrary behaviors.

2. DFD stores, which are often seen as abstractions for files, can be modeled using only data flows that hold a single (possibly compound) value. This also provides a model for shared variables in a straightforward manner. (A variation on this model that used flows that hold queues of values might provide a useful model of message passing in distributed systems.)

3. One way to model non-primitive bubbles (DFD processes) in hierarchically decomposed DFDs is to allow a bubble to fire concurrently with itself. In a variation of our basic model, a bubble may read again before it has written its output. This allows a bubble to act like a system of interacting bubbles, and thus a single bubble can have the dynamic behavior of a system of bubbles in a hierarchical DFD.

The rest of the paper describes our semantics for our theoretical core DFDs. (Thus, when we say "DFD" below we usually mean our core theoretical variant of the DFD notion, not the traditional notion.) It is organized as follows. Section 2 below describes the structure of DFDs. Section 3 describes configurations of DFDs and which are the basis of the operational semantics. Section 4 describes the meaning of a P-spec. Section 5 presents the basic operational semantics. Section 6 formalizes the notion of stores and discusses the implications of both persistent and consumable flows in our formalization of stores. Section 7 generalizes the way in which a bubble fires to include the possibility of a bubble firing concurrently with itself and highlights the possible use of this semantic generalization for hierarchically structured (refined) DFDs. Following that, Section 8 presents some discussion, including a discussion of related work.

2 Structure of a Data Flow Diagram

For the purposes of this paper, a data flow diagram (DFD) is modeled as a set of bubbles, B, and a set of labeled flows, F. (We do not consider hierarchical DFDs until Section 7. Imagine that the translation into our DFD semantics involves expansion of the hierarchical DFD first.) Bubbles are abstractions of processes, and are often written as circles, ovals, or boxes. Flows are abstractions of information movement or variables and are written as arrows. Thus the abstract syntax of a DFD is a labeled, directed graph, where the labeled arrows are the graph's directed edges, and the bubbles are the graph's nodes.

An example DFD for an accounts receivable system is given in Figure 1. (We will explain our strange treatment of stores below.) Informally, the idea behind this simplistic DFD is that when a customer communicates a need to a member of the sales staff, a master list of customer information is updated, and either a credit order or a cash order is generated. Both kinds of orders generate changes in the accounts receivable database, but credit orders also result in a billing. In the process of billing, a clerk sends a bill to the customer through the mail, and a copy of the bill is sent to update the accounts receivable database. A customer may also send a payment, consisting of money and the bills that are being paid. The credit from the payment to be applied to each account is used to update the accounts receivable database; that is, the collections process sorts out how much of the payment is to be applied to each account.

A named bubble in the graphical notation is represented by its bubble name in the formal model. So the set of bubbles, B, for the DFD in Figure 1 contains just the following names: `Customers`, `Generate-Sales-Order`, `Bill-Customer`, `Keep-Customer-Accounts`, `Collections`, and `Mail`.

The terminators (external entities) of a DFD are bubbles that are its sources and sinks, when the DFD is considered as a directed graph. In our example, boxes are used instead of ovals to denote terminators, and hence the bubbles `Customers` and `Mail` are its terminators. (Considering terminators to be part of the DFD may strike some as unusual, but it will be seen that they can be treated as bubbles, although in the usual case no interesting constraints will be specified on their behavior.)

In our graphical notation for a DFD, flows are labeled with three pieces of information: either a single or double arrowhead, a flow name $fn \in FLOWNAMES$, and a type name $T \in TYPES$. Neither the flow name nor the type alone are sufficient to uniquely identify a flow in our example; having both labels eliminates a common source of ambiguity [Col91].[1]

In SA, DeMarco [DeM78] and others (e.g., [War86]) make a distinction between "discrete" and "continuous" flows. However, some authors seem to consider continuous flows as continuous in the sense of calculus (i.e., real-valued, differentiable). DeMarco and Ward seem to think of them as continuously existing (i.e., like a shared variable) even when read. We adopt this latter interpretation, but to avoid confusion, we use different terminology [Col91]. Flows labeled with a double arrowhead are called persistent flows, which can be considered to be shared variables; flows with a single arrowhead are a consumable flows, which can be considered to be unbounded FIFO queues. For example, the flow from `Collections` to itself is a persistent flow; its name is `log`, and its type is `Money-and-Acct-List`.

Thus a flow is formally modeled by a 5-tuple, (b_1, fn, T, b_2, p), which represents a flow

[1] For textual representations, a better label is: a flow name $fn \in FLOWNAMES$, a source bubble name $b_{source} \in B$, and a target bubble name $b_{target} \in B$. This allows multiple flows with the same name and type and uniquely specifies the flow a bubble reads from and writes to, provided that flows with the same name and type are not allowed to have a common source and target bubble.

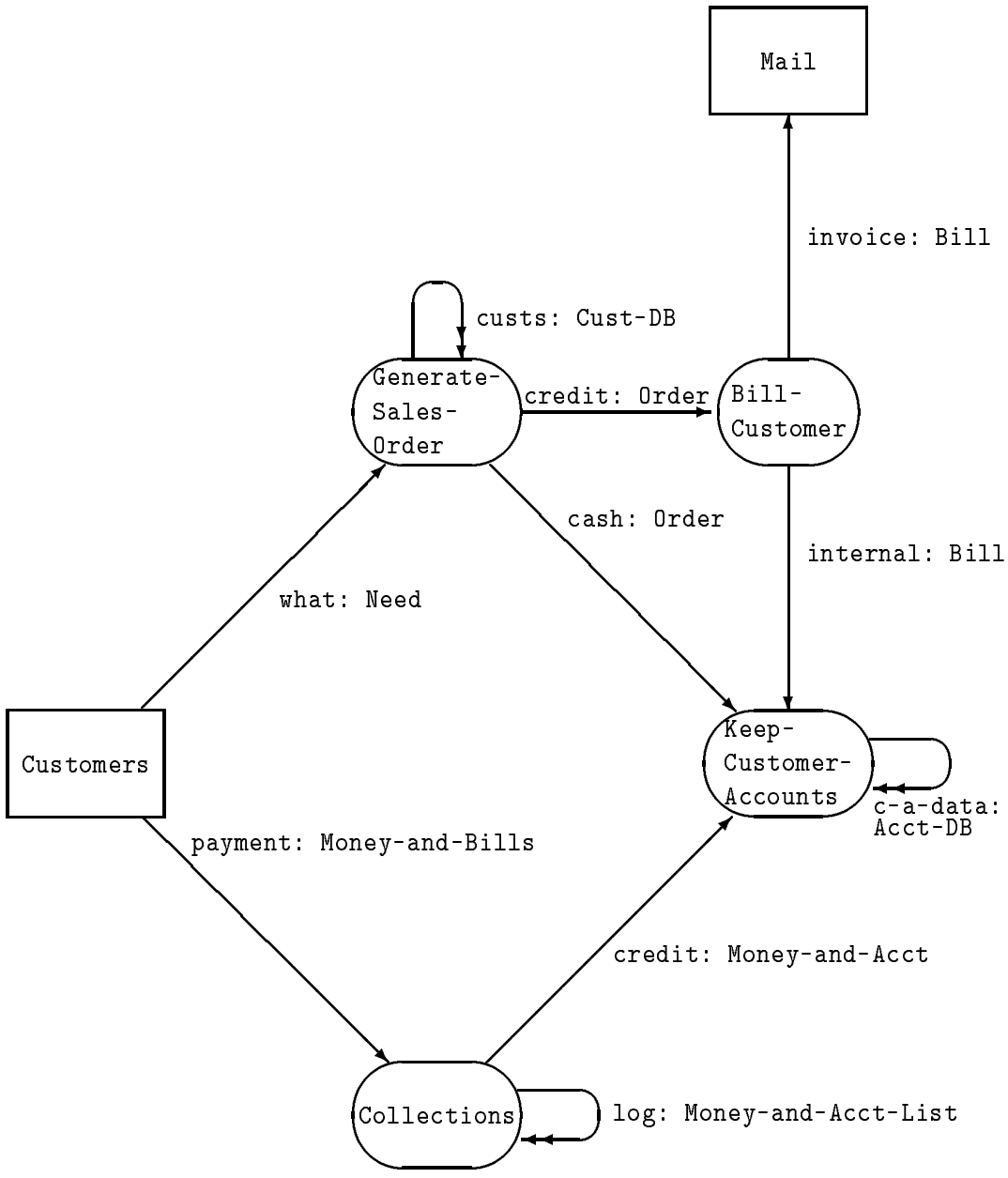

Figure 1: (Extended) data flow diagram of an accounts receivable system.

$f_{\texttt{c-a-data}}$ = (`Keep-Customer-Accounts`, `c-a-data`, `Acct-DB`, `Keep-Customer-Accounts`, *persistent*)
$f_{\texttt{cash}}$ = (`Generate-Sales-Order`, `cash`, `Order`, `Keep-Customer-Accounts`, *consumable*)
$f_{\texttt{creditBC}}$ = (`Generate-Sales-Order`, `credit`, `Order`, `Bill-Customer`, *consumable*)
$f_{\texttt{creditKCA}}$ = (`Collections`, `credit`, `Money-and-Acct`, `Keep-Customer-Accounts`, *consumable*)
$f_{\texttt{custs}}$ = (`Generate-Sales-Order`, `custs`, `Cust-DB`, `Generate-Sales-Order`, *persistent*)
$f_{\texttt{internal}}$ = (`Bill-Customer`, `internal`, `Bill`, `Keep-Customer-Accounts`, *consumable*)
$f_{\texttt{invoice}}$ = (`Bill-Customer`, `invoice`, `Bill`, `Mail`, *consumable*)
$f_{\texttt{payment}}$ = (`Customers`, `payment`, `Money-and-Bills`, `Collections`, *consumable*)
$f_{\texttt{log}}$ = (`Collections`, `log`, `Money-and-Acct-List`, `Collections`, *persistent*)
$f_{\texttt{what}}$ = (`Customers`, `what`, `Need`, `Generate-Sales-Order`, *consumable*)

Table 1: Abbreviations for the flows in Figure 1.

Notation for Members $\in$ Name	= description
$b \in B$	= a set (of bubble names)
$fn \in FLOWNAMES$	= a set (of flow names)
$T \in TYPES$	= a set (of type names)
$p \in P$	= {*persistent*, *consumable*}
$f \in F$	= $B \times FLOWNAMES \times TYPES \times B \times P$

Table 2: Domains describing the structure of a DFD.

named fn going from the bubble b_1 (the source) to the bubble b_2 (the target), carrying information of type T; this flow is persistent if p = *persistent* and consumable if p = *consumable*. For example, the flow from `Collections` to itself is formally described as the 5-tuple (`Collections`, `log`, `Money-and-Acct-List`, `Collections`, *persistent*). Since such 5-tuples tend to be rather unwieldy, for all the flows in Figure 1 we give abbreviations in Table 1. Abbreviations are based on the flow name, and the two flows named `credit` are distinguished by appending an abbreviation for their targets.

In a persistent flow, the shared variable that the flow represents can be written by the bubble at the source of the flow, and read by the target bubble [Col91]. Reading from a persistent flow does not change the information in the flow. The only persistent flows in our example go from bubbles to themselves; we use them to model stores. For example, $f_{\texttt{custs}}$ is used to model the store that would be attached to the bubble `Generate-Sales-Order`.

In a consumable flow, the source bubble enters tokens of information at the tail of the queue that the flow represents, and the target bubble removes information tokens from the head of the queue. (The dynamic behavior of flows is discussed in detail below.)

The model of a DFD's structure is summarized in Table 2.

2.1 Auxiliary Functions for Flow Components

It will be convenient to have some auxiliary functions for accessing the components of a flow. If $f = (b_1, fn, T, b_2, p)$, then $Source(f) = b_1$, $FlowName(f) = fn$, $TypeOf(f) = T$, and $Target(f) = b_2$. The value of $Consumable(f)$ is *true* if p = *consumable* and is

Function :	Type
$Source$:	$F \rightarrow B$
$FlowName$:	$F \rightarrow FLOWNAMES$
$TypeOf$:	$F \rightarrow TYPES$
$Target$:	$F \rightarrow B$
$Consumable$:	$F \rightarrow Boolean$
$Inputs$:	$B \rightarrow PowerSet(F)$
$Outputs$:	$B \rightarrow PowerSet(F)$
$TypeMeaning$:	$TYPES \rightarrow Set$

Table 3: Auxiliary Functions for describing the structure of a DFD.

false otherwise. For example, if $f_{\texttt{payment}}$ is the flow from Figure 1 described in Table 1, then $Source(f_{\texttt{payment}}) = \texttt{Customers}$, $FlowName(f_{\texttt{payment}}) = \texttt{payment}$, $TypeOf(f_{\texttt{payment}}) = \texttt{Money-and-Bills}$, $Target(f_{\texttt{payment}}) = \texttt{Collections}$, and $Consumable(f_{\texttt{payment}}) = true$.

A flow f is said to be an *input flow* of a bubble b if $Target(f) = b$. We write $Inputs(b)$ for the set of all of b's input flows. Similarly, f is an *output flow* of b if $Source(f) = b$. We write $Outputs(b)$ for the set of all of b's output flows. In our example, $Inputs(\texttt{Mail}) = \{f_{\texttt{invoice}}\}$ and $Outputs(\texttt{Mail}) = \{\}$.

2.2 Meanings of Types

A "data dictionary" is often associated with a DFD, and sometimes describes all the flows. However, for our purposes, we consider the data dictionary to define the meaning of all the types used in a DFD. In what follows we will need very little of the semantics of types; so it will suffice to think of a type as describing a set of objects. The set of objects associated with a type T is given by $TypeMeaning(T)$.

The above information is summarized in Table 3. In this table, by *Set* we mean the class of all recursive sets. This suffices for normal use of DFDs, but more exotic domains could be used to model types such as higher-order functions or lazy streams.

3 Configurations of Data Flow Diagrams

The execution of a DFD consists of bubble computations and information moving from one bubble to another along the flows. We call a snapshot at certain well-defined points of such an execution a configuration; it is similar to a marking of a Petri net [Pet77]. A configuration records the mode of each bubble (see below), the information that bubble has read from its input flows (if any), and the information present on each flow.

3.1 Bubble Modes

We use a two-step firing rule for bubbles in order to model concurrent firing of bubbles and time delays. If bubbles were able to read their inputs and compute and write their outputs in one atomic step, then we would be unable to model time delays and race conditions that can occur in systems, where processes take a finite amount of time to compute results.

For example, consider the DFD shown in Figure 2. Suppose that the bubble `Test` periodically changes the value on the persistent flows named `y` and `z` by setting both to the

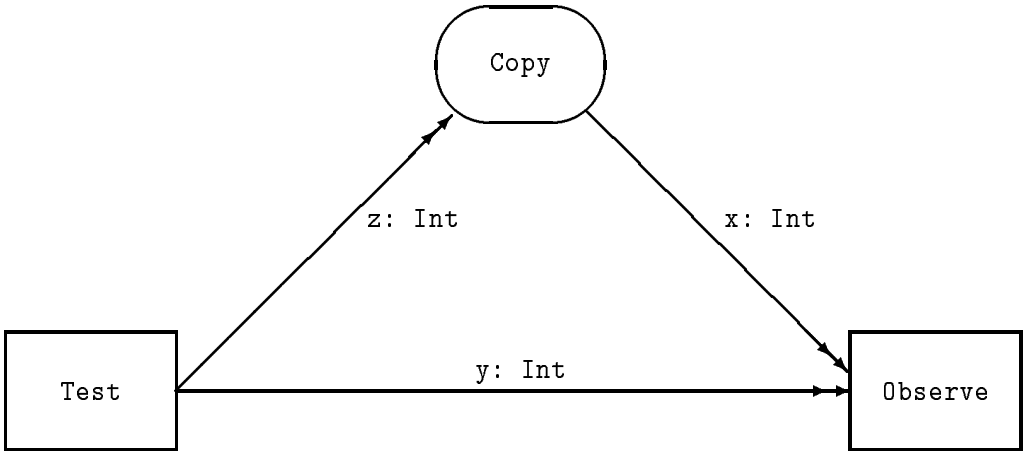

Figure 2: Data flow diagram showing a race condition.

same value in the sequence $0, 1, 2, 3, \ldots$. Suppose that the bubble `Copy` copys its input on `z` to `x`, and that the initial values of `x`, `y`, and `z` are 0. Finally, observe the results through the bubble `Observe`; that is observe a sequence of pairs of the form $(\mathtt{x} = x_i, \mathtt{y} = y_i)$. Under an atomic firing rule, if we observe a pair $(\mathtt{x} = x_i, \mathtt{y} = y_i)$, then the next pair we observe, $(\mathtt{x} = x_{i+1}, \mathtt{y} = y_{i+1})$, must satisfy one of the following conditions:

- $x_{i+1} = x_i$.

  This will be the case if `Copy` has *not* fired, since only `Copy` writes the flow named `x`.

- $x_{i+1} \geq y_i$.

  This will be the case if `Copy` fired. It is impossible for x_{i+1} to be less than y_i (the previous value on `y` and `z`), because the value on `y` and `z` only increases, and if `Copy` fires it atomically copies to `x` the value on `y` and `z`, which must be at least y_i.

The behavior of the DFD in Figure 2 is different if `Copy` can fire in two steps, first by reading its inputs, and then later by writing its output. To be concrete, consider the initial configuration (where `x`, `y`, and `z` are 0). In this configuration one can observe the pair $(\mathtt{x} = 0, \mathtt{y} = 0)$. Suppose that `Test` changes `y` and `z` to 1. Now suppose `Copy` reads the value `1` from `z`. Suppose that `Test` changes `y` and `z` to 2. In this configuration one can observe the pair $(\mathtt{x} = 0, \mathtt{y} = 2)$. Finally, suppose that `Copy` writes out the value it read; that is, it sets `x` to 1. In this configuration one can observe the pair $(\mathtt{x} = 1, \mathtt{y} = 2)$. The sequence of pairs observed in this execution would be impossible with the atomic firing rule, because the value of `x`, 1, is neither the previous value of `x`, 0, nor is it greater than or equal to the previous value of `y` (and `z`), 2. We emphasize that this sequence of observations would be possible in a world where bubbles take a finite time to execute and where executions can overlap. Hence the necessity of non-atomic firings in an adequate formal model.

To allow non-atomic firings, each bubble can be in one of two modes: *idle*, which means that the bubble has not read its input flows, or *working*, which means that the bubble has read its input flows (and is computing). The modes of the bubbles in a DFD configuration are represented by a total function from the set B of bubble names to modes. We use *BubbleMode* to stand for the set of all such functions. If $bm \in \mathit{BubbleMode}$ is such a function, then $bm(b)$ represents the mode of b. Changes in bubble modes are represented by changing to a new function in a new configuration.

As an example of a bubble mode function, consider the following, which uses the DFD of Figure 1. Suppose that bm_e is the function defined as follows.

$$bm_e(\texttt{Customers}) = \mathit{idle}$$
$$bm_e(\texttt{Generate-Sales-Order}) = \mathit{idle}$$
$$bm_e(\texttt{Bill-Customer}) = \mathit{working}$$
$$bm_e(\texttt{Keep-Customer-Accounts}) = \mathit{idle}$$
$$bm_e(\texttt{Collections}) = \mathit{working}$$
$$bm_e(\texttt{Mail}) = \mathit{idle}$$

Then this represents a state in which the bubbles `Bill-Customer` and `Collections` are working concurrently. (The accountants and sales force are on a coffee break.) In this form of the semantics, bubbles can fire concurrently with each other, but not concurrently with themselves. We relax this restriction in Section 7.

3.2 Information that a Bubble has Read from its Inputs

The information read by a bubble in its transition from *idle* to *working* cannot be discarded. This information is captured by curried functions from bubble names to functions from flows to the information the bubble has read from that flow. We use *Read* to stand for the set of all such functions. (To describe the domain *Read* in Table 4, we use a set, *OBJECTS*, that contains all types of objects that may appear on flows. We write $OBJECTS_\perp$ for the set $OBJECTS \cup \{\perp\}$. The notation $\perp$ means "no information."[2]) Suppose that $r \in Read$ is such a function, b is a bubble name, and f is a flow. Then $r(b)$ is of type *WhatRead* — a function from flow names to information, representing what b has read from its input flows. If the bubble b has not read any information from flow f, then the value of $r(b)(f)$[3] is $\perp$; otherwise the value of $r(b)(f)$ should be an element of the type of the flow f:

$$(r(b)(f) \neq \perp) \Rightarrow r(b)(f) \in \mathit{TypeMeaning}(\mathit{TypeOf}(f)). \tag{1}$$

For each bubble b, the mapping $r(b)$, where $r \in Read$, should only be defined on flows that are inputs to b. This is stated formally as follows. For each $r \in Read$, for each $b \in B$, for each $f \in F$:

$$(f \notin \mathit{Inputs}(b)) \Rightarrow (r(b)(f) = \perp). \tag{2}$$

An example of a read function is the following, where $co \in \mathit{TypeMeaning}(\texttt{Order})$, $pmt \in \mathit{TypeMeaning}(\texttt{Money-and-Bills})$, and $() \in \mathit{TypeMeaning}(\texttt{Money-and-Acct-List})$.[4]

$r_e(\texttt{Customers}) = \lambda f\,.\,\perp$
$r_e(\texttt{Generate-Sales-Order}) = \lambda f\,.\,\perp$
$r_e(\texttt{Bill-Customer}) = \lambda f\,.\,\textbf{if } f = f_{\texttt{creditBC}} \textbf{ then } co \textbf{ else } \perp \textbf{ fi}$
$r_e(\texttt{Keep-Customer-Accounts}) = \lambda f\,.\,\perp$
$r_e(\texttt{Collections}) = \lambda f\,.\,\textbf{if } f = f_{\texttt{payment}} \textbf{ then } pmt \textbf{ else if } f = f_{\texttt{log}} \textbf{ then } () \textbf{ else } \perp \textbf{ fi fi}$
$r_e(\texttt{Mail}) = \lambda f\,.\,\perp$

[2] We also use $\perp$ to indicate that a bubble read from an empty flow. This should not happen if the enabling rules of a bubble have been properly constructed, so we do not provide a way to distinguish the two ways to not read information from a flow.

[3] The notation $r(b)(f)$ means the result of $r(b)$ applied to f. Function application is left associative. We ask readers familiar with such notational conventions to bear with us as we try to make this paper accessible to a wider audience.

[4] The notation $\lambda f\,.\,e$ denotes a function that takes an argument, and returns the value of the expression e with the argument substituted for f [Chu41] [Sch86].

The function r_e describes part of a configuration where the bubble `Bill-Customer` has read the order *co* from the flow $f_{\texttt{creditBC}}$ (see Table 1), and where `Collections` has read the payment *pmt* from $f_{\texttt{payment}}$ and an empty list from $f_{\texttt{log}}$.

3.3 Information on a Flow

The information on the flows of a DFD is captured by a function from flows to a finite sequence of objects of the appropriate type. A persistent flow must be mapped to a sequence of length zero or one; when the sequence is empty, it represents an uninitialized persistent flow. (An alternative would be to use $\perp$ for uninitialized persistent flows.) We use *FlowState* to stand for the set of all such functions *fs*. Formally an element $fs \in FlowState$ must be such that the following conditions are satisfied for all $f \in F$:

- $TypeOf(fs(f)) = TypeOf(f)^*$, and
- $\neg Consumable(f) \Rightarrow length(fs(f)) = 1$,

where *length* is the usual length function on sequences.

For example, the function fs_e such that:

$$fs_e(f) = \begin{cases} \langle cdb \rangle & \text{if } f = f_{\texttt{custs}} \\ \langle ardb \rangle & \text{if } f = f_{\texttt{c-a-data}} \\ \langle () \rangle & \text{if } f = f_{\texttt{log}} \\ \langle \rangle & \text{otherwise,} \end{cases}$$

describes part of a configuration where there are no values on any of the consumable flows, and various values (*cdb*, *ardb*, and ()) on the persistent flows of the DFD in Figure 1.

3.4 Auxiliary Functions on Sequences

We treat sequences of objects, that is elements of $(OBJECTS^*)_\perp$, as FIFO queues using the following constants and operations.

$$\begin{array}{rcl} \langle\rangle & : & OBJECTS^* \\ Enq & : & (OBJECTS^*)_\perp \times OBJECTS \to (OBJECTS^*)_\perp \\ IsEmpty & : & (OBJECTS^*)_\perp \to Boolean_\perp \\ Head & : & (OBJECTS^*)_\perp \to OBJECTS_\perp \\ Rest & : & (OBJECTS^*)_\perp \to (OBJECTS^*)_\perp \end{array}$$

These operations are defined to satisfy the following equations for all $q \in OBJECTS^*$ and $o \in OBJECTS$.

$$\begin{array}{rcl} Enq(\perp, o) & = & \perp \\ IsEmpty(\perp) & = & \perp \\ IsEmpty(\langle\rangle) & = & true \\ IsEmpty(Enq(q, o)) & = & false \\ Head(\perp) & = & \perp \\ Head(\langle\rangle) & = & \perp \\ Head(Enq(q, o)) & = & \textbf{if } IsEmpty(q) \textbf{ then } o \textbf{ else } Head(q) \textbf{ fi} \\ Rest(\perp) & = & \perp \\ Rest(\langle\rangle) & = & \perp \\ Rest(Enq(q, o)) & = & \textbf{if } IsEmpty(q) \textbf{ then } \langle\rangle \textbf{ else } Enq(Rest(q), o) \textbf{ fi} \end{array}$$

Notation for Members $\in$ Name	= description
$m \in MODES$	$= \{idle, working\}$
$bm \in BubbleMode$	$= B \to MODES$
$o \in OBJECTS$	$= \bigcup_{T \in TYPES} TypeMeaning(T)$
$s \in WhatRead$	$= (F \to OBJECTS_\perp)$
$r \in Read$	$= B \to WhatRead$
$fs \in FlowState$	$= F \to OBJECTS^*$
$\gamma \in$,	$= BubbleMode \times Read \times FlowState$

Table 4: Domains describing the configuration of a DFD.

In dealing with $\perp$ in example formulas, we assume that all boolean functions are strict; for example, $(\neg\perp) = \perp$ and $(true \vee \perp) = \perp$.

3.5 Configurations

A configuration summarizes all the information about the state of a DFD described above.

Formally, a *configuration* of a DFD is a triple, (bm, r, fs), where $bm \in BubbleMode$, $r \in Read$, and $fs \in FlowState$. The set of all configurations is denoted by , . This is summarized in Table 4.

An example configuration of the DFD in Figure 1 is the triple (bm_e, r_e, fs_e) where bm_e, r_e and fs_e are as described above.

3.6 Discussion

We do not model "control flows" explicitly. One can translate a diagram that has a control flow from b_1 to b_2 into a DFD with a flow $(b_1, fn, Signal, b_2, consumable)$, where *Signal* is a type that has only one element. (One could also use a persistent flow and a two-element type to indicate that the signal is "on" or "off".)

We defer further discussion of how to model the stores of a DFD until Section 6.

4 Meaning of a P-Spec

The things that the bubbles in a DFD can do are specified by a *P-spec*. A P-spec may be presented in many different forms; for example, some authors use finite state machines, and others use a formal specification language such as VDM or Z. Because our model is intended to be a target for any such specification language, we must be more general. Thus we adopt a formal model of P-specs that consists of three curried functions and an initial *FlowState* map:

$$(Enabled, Consume, Produce, fs_{\text{initial}}).$$

We do not show how to translate a P-spec into these functions, as this depends on the technique used in presenting the P-spec. The three curried functions tell when a bubble in the DFD is enabled, what it consumes when it is enabled, and what it produces when it is finished working. We allow bubbles to be nondeterministic in what they consume and produce, as this is sometimes convenient in specifications.

4.1 When are Bubbles Enabled?

In a given flow state, a bubble is enabled if it would be able to go from *idle* to *working*. A bubble's being enabled depends on the flow state, because if the bubble needs to read consumable inputs, there must be a non-empty sequence of inputs available on the needed flows.

In our formal model, we allow enablement to depend on both the presence of values on input flows as well as on the values on such flows. Some specification languages might not allow enablement to depend on the values on flows, the model supports those that do.

The part of P-spec that tells what bubbles are enabled is captured by a mapping:

$$Enabled : B \rightarrow (FlowState \rightarrow Boolean_{\perp}).$$

This function is curried, so that for a bubble b, $Enabled(b)$ tells the flow states in which b is enabled. In a given flow state, fs, there may be no information on a given flow; this is why $Enabled(b)(fs)$ is allowed to be $\perp$; a bubble b is only considered enabled in fs if $Enabled(b)(fs) = true$. The semantics only asks if a bubble is enabled if it is in *idle* mode.

The mapping $Enabled(b)$ should only depend on the states of the flows in $Inputs(b)$; formally, this condition is stated as follows. For all $fs \in FlowState$ and $fs' \in FlowState$:

$$(\forall f \in Inputs(b) : fs(f) = fs'(f)) \Rightarrow Enabled(b)(fs) = Enabled(b)(fs'). \quad (3)$$

As an example, the mapping $Enabled_e$ defined as follows would be suitable for the DFD of Figure 1.

$$Enabled_e(\texttt{Customers}) = \lambda fs\ .\ true$$
$$Enabled_e(\texttt{Generate-Sales-Order}) = \lambda fs\ .\ \neg IsEmpty(fs(f_{\texttt{what}}))$$
$$Enabled_e(\texttt{Bill-Customer}) = \lambda fs\ .\ \neg IsEmpty(fs(f_{\texttt{creditBC}}))$$
$$Enabled_e(\texttt{Keep-Customer-Accounts}) = \lambda fs\ .$$
$$\neg IsEmpty(fs(f_{\texttt{cash}})) \vee \neg IsEmpty(fs(f_{\texttt{internal}})) \vee \neg IsEmpty(fs(f_{\texttt{creditKCA}}))$$
$$Enabled_e(\texttt{Collections}) = \lambda fs\ .\ \neg IsEmpty(fs(f_{\texttt{payment}}))$$
$$Enabled_e(\texttt{Mail}) = \lambda fs\ .\ \neg IsEmpty(fs(f_{\texttt{invoice}}))$$

This says, for example, that `Keep-Customer-Accounts` is enabled if there is something on one of the three consumable flows into that bubble. Similarly, `Collections` is enabled if there is something on the flow named `payment`.

4.2 What do Bubbles Consume when they are Enabled?

When a bubble is enabled, its mode may be changed from *idle* to *working*. At this point it reads some of its input flows, and may consume some of these. Only consumable flows may be consumed, and consumption means removing the head of the sequence associated with the flow. (The information read from these consumable flows is saved in the next configuration's *Read* map by the firing rules below.)

The part of a P-spec that says what *idle* bubbles will consume in a given flow state if they make the transition from *idle* to *working* mode is captured in the curried, set-valued mapping:

$$Consume : B \rightarrow ((FlowState \times Read) \rightarrow PowerSet(FlowState \times Read)).$$

That is, for all *idle* bubbles b, and all pairs (fs, r) of *FlowState* and *Read* mappings, the set $Consume(b)(fs, r)$ is a set of pairs of *FlowState* and *Read* mappings. Because *Consume* is

The following must hold for all $b \in B$, $fs \in \mathit{FlowState}$, and $r \in \mathit{Read}$.

[c-range:] If $\mathit{Enabled}(b)(fs) = \mathit{true}$, then $\mathit{Consume}(b)(fs, r) \neq \emptyset$.

[c-local:] If $(fs', r') \in \mathit{Consume}(b)(fs, r)$, then

- for all $b' \neq b$, $r'(b') = r(b')$ and
- for all $f \in F$:
  - if $f \notin \mathit{Inputs}(b)$ then $fs'(f) = fs(f)$, and
  - if $f \in \mathit{Inputs}(b)$ then
    - if $\mathit{Consumable}(f)$, then either:
      - $fs'(f) = fs(f)$ and $r'(b)(f) = \perp$, or
      - $\mathit{IsEmpty}(fs(f)) = \mathit{false}$ and $fs'(f) = \mathit{Rest}(fs(f))$ and $r'(b)(f) = \mathit{Head}(fs(f))$,
    - if $\neg\mathit{Consumable}(f)$, then $fs'(f) = fs(f)$ and either:
      - $r'(b)(f) = \perp$, or
      - $\mathit{IsEmpty}(fs(f)) = \mathit{false}$ and $r'(b)(f) = \mathit{Head}(fs(f))$.

Figure 3: Restrictions on *Consume*.

curried, for a bubble b, $\mathit{Consume}(b)$ is the mapping derived from b's P-spec. The mapping $\mathit{Consume}(b)$ can also be thought of as a binary relation between pairs of *FlowState* and *Read* mappings. A relation or set-valued mapping is needed to deal with possible nondeterminism in the P-spec.

Each *FlowState-Read* pair in the set $\mathit{Consume}(b)(fs, r)$ represents a possible change in fs and r that may occur when the bubble b makes the transition from *idle* to *working*. The set of possible changes, $\mathit{Consume}(b)(fs, r)$, is assumed to be non-empty when $\mathit{Enabled}(b)(fs)$ is *true*. The changes should be local to the bubble b; that is, each *FlowState-Read* pair in $\mathit{Consume}(b)(fs, r)$ should only differ from (fs, r) on the input flows of b and in the mapping $r(b)$. A consumable flow need not be consumed, but if it is consumed, the head of the flow is taken off and read by the bubble. A persistent flow cannot be consumed, but it may be read. A flow can be read only if it is defined and non-empty. The above restrictions are stated formally in Figure 3.

To construct a *Consume* mapping for our example, we first introduce two notations.

A notation for updating mappings at a given point is helpful, because so much of the "state" of a DFD is encoded as functions. The notation $[x \mapsto y]g$ is an update to a function, g; it is defined by the following equation.

$$[x \mapsto y]g \stackrel{\text{def}}{=} \lambda z \,.\, \textbf{if } z = x \textbf{ then } y \textbf{ else } g(z) \textbf{ fi} \tag{4}$$

For example, $[f \mapsto \mathit{Rest}(fs(f))]fs$ is the function that is just like fs, except that the head of the sequence on f has been removed.

The function *In*, defined below, will consume a given input flow for a given bubble. It is a curried function, so $\mathit{In}(f, b)$ represents just the changes that b makes to the flow state

and read function by reading the flow f. It only changes the flow state when the flow is consumable. It is formally defined as follows.[5]

$$
\begin{array}{l}
In : (F \times B) \to ((\mathit{FlowState} \times \mathit{Read}) \to (\mathit{FlowState} \times \mathit{Read})) \\
In(f, b)(fs, r) = \\
\quad \textbf{let } r_b = [f \mapsto \mathit{Head}(fs(f))](r(b)) \\
\quad \textbf{in } (\textbf{if } \mathit{Consumable}(f) \textbf{ then } [f \mapsto \mathit{Rest}(fs(f))]fs \textbf{ else } fs \textbf{ fi}, \\
\qquad [b \mapsto r_b]r)
\end{array}
$$

(Updating the read function of a configuration when a bubble consumes some of its inputs causes no problems, since the read function originally maps each of that bubble's input flows to $\perp$.)

As an example, the mapping $\mathit{Consume}_e$ defined as follows would be suitable for the DFD of Figure 1. The clause for `Generate-Sales-Order` says that there is only one *FlowState*, *Read* pair possible, that only the flow named `what` is consumed (because the flow named `custs` is persistent), and that both of the input flows are read.

$$
\begin{array}{l}
\mathit{Consume}_e(\texttt{Customers}) = \lambda(fs, r)\,.\,\{(fs, r)\} \\
\mathit{Consume}_e(\texttt{Generate-Sales-Order}) = \lambda(fs, r)\,. \\
\quad \{\mathit{In}(f_{\texttt{what}}, \texttt{Generate-Sales-Order})(\\
\qquad \mathit{In}(f_{\texttt{custs}}, \texttt{Generate-Sales-Order})(fs, r))\} \\
\mathit{Consume}_e(\texttt{Bill-Customer}) = \lambda(fs, r)\,.\,\{\mathit{In}(f_{\texttt{creditBC}}, \texttt{Bill-Customer})(fs, r)\} \\
\mathit{Consume}_e(\texttt{Keep-Customer-Accounts}) = \lambda(fs, r)\,. \\
\quad \{\mathit{In}(f_{\texttt{c-a-data}}, \texttt{Keep-Customer-Accounts})(\\
\qquad \mathit{In}(f_{\texttt{cash}}, \texttt{Keep-Customer-Accounts})(fs, r)), \\
\quad \mathit{In}(f_{\texttt{c-a-data}}, \texttt{Keep-Customer-Accounts})(\\
\qquad \mathit{In}(f_{\texttt{internal}}, \texttt{Keep-Customer-Accounts})(fs, r)), \\
\quad \mathit{In}(f_{\texttt{c-a-data}}, \texttt{Keep-Customer-Accounts})(\\
\qquad \mathit{In}(f_{\texttt{creditKCA}}, \texttt{Keep-Customer-Accounts})(fs, r))\} \\
\mathit{Consume}_e(\texttt{Collections}) = \lambda(fs, r)\,. \\
\quad \{\mathit{In}(f_{\texttt{log}}, \texttt{Collections})(\mathit{In}(f_{\texttt{payment}}, \texttt{Collections})(fs, r)\} \\
\mathit{Consume}_e(\texttt{Mail}) = \lambda(fs, r)\,.\,\{\mathit{In}(f_{\texttt{invoice}}, \texttt{Mail})(fs, r)\}
\end{array}
$$

One can check that the conditions of Figure 3 are satisfied by $\mathit{Consume}_e$ by cases, that is, bubble by bubble. For example, consider the bubble `Customers`, which is always enabled. Because this bubble is always enabled, $\mathit{Consume}_e(\texttt{Customers})$ always produces a non-empty set, and thus satisfies the condition **c-range**. For this bubble, condition **c-local** is trivially satisfied, because the only pair in the set returned by $\mathit{Consume}_e(\texttt{Customers})$ is its argument.

Another way to check that the conditions of Figure 3 are satisfied by $\mathit{Consume}_e$ is to check **c-range** and **c-local** separately. It is easy to check **c-range**, since for all bubbles $b \in B$, $\mathit{Consume}_e(b)(fs, r) \neq \emptyset$ by construction. (Closer comparison of $\mathit{Consume}_e$ with $\mathit{Enabled}_e$ is necessary to show that $\mathit{Consume}_e$ does the right thing when enabled, but that is another story.)

Checking **c-local** can be based on the definition of In, used in its construction. For each bubble b, if $f \in \mathit{Inputs}(b)$, then $\lambda(fs, r)\,.\,\{\mathit{In}(f, b)(fs, r)\}$ satisfies **c-local** for b by construction; that is, In satisfies the conditions of **c-local** for persistent and consumable

[5] What In returns is a pair, as noted by its type. The pair follows the **in** of the **let in** notation [Sch86], with the first element of the pair before the comma, and the second after.

flows. Since the result of one application of $In(f, b)$ satisfies **c-local**, and since $In(f', b)$ leaves the flow state and read state for f alone when $f' \neq f$, it follows that compositions of In of the form $\lambda(fs, r)\ .\ \{In(f', b)(In(f, b)(fs, r))\}$ also satisfy **c-local**, if $f' \neq f$ and $f' \in Inputs(b)$. Unions of sets representing functions that satisfy **c-local** also satisfy **c-local**, so alternatives as in $Consume_e$(`Keep-Customer-Accounts`) can be checked separately.

4.3 What do Bubbles Produce when they Finish their Work?

Each bubble in *working* mode can be called on to produce some output in the transition from *working* to *idle*. This transition is modeled by a mapping, *Produce*, which is similar to the *Consume* mapping above:

$$Produce : B \rightarrow ((FlowState \times Read) \rightarrow PowerSet(FlowState \times Read)).$$

The idea is that for all bubbles b, and all pairs (fs, r) of *FlowState* and *Read* mappings, the set $Produce(b)(fs, r)$ is a set of pairs of *FlowState* and *Read* mappings. Again, *Produce* is curried, so that for a bubble b, $Produce(b)$ is the is the mapping derived from b's P-spec. The semantics only ask what a bubble produces when it is in *working* mode.

Each *FlowState-Read* pair in the set $Produce(b)(fs, r)$ represents a possible change in fs and r that could be caused by b's transition from *working* to *idle*. This set must be non-empty, so that the bubble may always go from *working* mode to *idle* mode. The changes represented by $Produce(b)(fs, r)$ should be local to the bubble b. Furthermore, nothing about what was read is remembered in the new *Read* mapping after producing output. A consumable output flow need not be changed, but if it is, a single value is produced and added to the tail of that flow's queue. A persistent flow also need not be changed, but if it is, then the new value replaces the old value on the flow.

There is an additional locality condition needed for $Produce(b)$. This condition ensures that what a bubble produces only depends on what it has read. More precisely, both the values that a bubble b produces and the set of choices it offers should not depend on anything except $r(b)$.

The above restrictions are formally stated in Figure 4.

To work an example, we define a function, similar to In, called Out. The function Out, defined below, will output a given object on a given flow for a given bubble. It is a curried function, so $Out(o, f, b)$ represents just the changes that b makes to the flow state and read function by producing o on the flow f.

$Out : (OBJECTS \times F \times B) \rightarrow ((FlowState \times Read) \rightarrow (FlowState \times Read))$
$Out(o, f, b)(fs, r) =$
  let $r_b = \lambda f'\ .\ \bot$
  in (**if** $Consumable(f)$ **then** $[f \mapsto Enq(fs(f), o)]fs$ **else** $[f \mapsto Enq(\langle\rangle, o)]fs$ **fi**,
    $[b \mapsto r_b]r$)

A *Produce* mapping for the diagram in Figure 1 is as follows. For this example, assume that the functions *needsCredit*, *makeCashOrder*, *makeCreditOrder*, *updateCashCust*, *updateCreditCust*, *makeBill*, *updateAccts*, *creditOfPayment*, and *updateLog* are defined elsewhere (e.g., in a P-spec). Because the bubble `Customers` is a source, we know that it can produce output, but have no way of knowing what that output might be. Thus, the *Produce* mapping given allows `Customers` to have any behavior that is consistent with the types of its outflows. The sink `Mail` is given a *Produce* mapping that throws away the information it has read, and produces no output.

The following must hold for all $b \in B$, $fs \in \mathit{FlowState}$, and $r \in \mathit{Read}$.

[p-range:] $\mathit{Produce}(b)(fs, r) \neq \emptyset$.

[p-local:] If $(fs', r') \in \mathit{Produce}(b)(fs, r)$, then

- for all $b' \neq b$, $r'(b) = r(b)$,
- for all $f \in F$, $r'(b)(f) = \perp$, and
- for all $f \in F$:
  - if $f \notin \mathit{Outputs}(b)$ then $fs'(f) = fs(f)$, and
  - if $f \in \mathit{Outputs}(b)$, then for some $o \in \mathit{TypeMeaning}(\mathit{TypeOf}(f))$:
    - if $\mathit{Consumable}(f)$, then either:
      - $fs'(f) = fs(f)$, or
      - $fs'(f) = \mathit{Enq}(fs(f), o)$.
    - if $\neg\mathit{Consumable}(f)$, then either:
      - $fs'(f) = fs(f)$, or
      - $fs'(f) = \mathit{Enq}(\langle\rangle, o)$.

[p-depend:] For all $fs' \in \mathit{FlowState}$, for all $r' \in \mathit{Read}$, if $r(b) = r'(b)$, then there is a one-to-one correspondence between $\mathit{Produce}(b)(fs, r)$ and $\mathit{Produce}(b)(fs', r')$ that sends each $(fs_0, r_0) \in \mathit{Produce}(b)(fs, r)$ to a $(fs_1, r_1) \in \mathit{Produce}(b)(fs', r')$ such that: for all $f \in \mathit{Outputs}(b)$ and for all $o \in \mathit{TypeMeaning}(\mathit{TypeOf}(f))$,

- $fs_0(f) \neq fs(f)$ if and only if $fs_1(f) \neq fs'(f)$, and
- either $fs_0(f) = \mathit{Enq}(fs(f), o)$ or $fs_0(f) = \mathit{Enq}(\langle\rangle, o)$ if and only if either $fs_1(f) = \mathit{Enq}(fs'(f), o)$ or $fs_1(f) = \mathit{Enq}(\langle\rangle, o)$.

Figure 4: Restrictions on *Produce*.

$$
\begin{aligned}
&\mathit{Produce}_e(\texttt{Customers}) = \lambda(fs, r)\,.\\
&\qquad \{\mathit{Out}(o, f_{\texttt{what}}, \texttt{Customers})(fs, r) \mid o \in \mathit{TypeMeaning}(\texttt{Need})\}\\
&\qquad \cup\{\mathit{Out}(o, f_{\texttt{payment}}, \texttt{Customers})(fs, r) \mid o \in \mathit{TypeMeaning}(\texttt{Money-and-Bills})\}\\
&\qquad \cup\{\mathit{Out}(o_1, f_{\texttt{what}}, \texttt{Customers})(\mathit{Out}(o_2, f_{\texttt{payment}}, \texttt{Customers})(fs, r)) \mid\\
&\qquad\qquad o_1 \in \mathit{TypeMeaning}(\texttt{Need}), o_2 \in \mathit{TypeMeaning}(\texttt{Money-and-Bills})\}\\
&\mathit{Produce}_e(\texttt{Generate-Sales-Order}) = \lambda(fs, r)\,.\\
&\qquad \textbf{if } \mathit{needsCredit}(r)\\
&\qquad \textbf{then } \{\mathit{Out}(\mathit{updateCreditCust}(r), f_{\texttt{custs}}, \texttt{Generate-Sales-Order})(\\
&\qquad\qquad \mathit{Out}(\mathit{makeCreditOrder}(r), f_{\texttt{creditBC}}, \texttt{Generate-Sales-Order})(fs, r))\}\\
&\qquad \textbf{else } \{\mathit{Out}(\mathit{updateCashCust}(r), f_{\texttt{custs}}, \texttt{Generate-Sales-Order})(\\
&\qquad\qquad \mathit{Out}(\mathit{makeCashOrder}(r), f_{\texttt{cash}}, \texttt{Generate-Sales-Order})(fs, r))\}\\
&\qquad \textbf{fi}\\
&\mathit{Produce}_e(\texttt{Bill-Customer}) = \lambda(fs, r)\,.\\
&\qquad \textbf{let } bl = \mathit{makeBill}(r) \textbf{ in}\\
&\qquad \{\mathit{Out}(bl, f_{\texttt{internal}}, \texttt{Bill-Customer})(\mathit{Out}(bl, f_{\texttt{invoice}}, \texttt{Bill-Customer})(fs, r))\}\\
&\mathit{Produce}_e(\texttt{Keep-Customer-Accounts}) = \lambda(fs, r)\,.\\
&\qquad \{\mathit{Out}(\mathit{updateAccts}(r), f_{\texttt{c-a-data}}, \texttt{Keep-Customer-Accounts})(fs, r)\}\\
&\mathit{Produce}_e(\texttt{Collections}) = \lambda(fs, r)\,.\\
&\qquad \{\mathit{Out}(\mathit{creditOfPayment}(r), f_{\texttt{credit}}, \texttt{Collections})(\\
&\qquad\qquad \mathit{Out}(\mathit{updateLog}(r), f_{\texttt{log}}, \texttt{Collections})(fs, r))\}\\
&\mathit{Produce}_e(\texttt{Mail}) = \lambda(fs, r)\,.\,\{(fs, [\texttt{Mail} \mapsto \lambda f\,.\,\bot]r)\}
\end{aligned}
$$

It is trivial to check that condition **p-range** of Figure 4 is satisfied by $\mathit{Produce}_e$. Checking **p-local** and **p-depend** can be based on the definition of Out, in the same way as we checked **c-local** for $\mathit{Consume}_e$; one must also assume that the definitions of all the auxiliary functions only consult the read state of the bubble in whose clause they appear. The details are left as an exercise for the reader.

5 Operational Semantics for Firing

We now have enough machinery to define the semantics of firing rules. In this (first) semantics, we do not allow a bubble to fire concurrently with itself (a condition we relax in Section 7). Thus there are just two kinds of transitions allowed between configurations: an enabled bubble can go from *idle* to *working*, and a *working* bubble can go to *idle*. Formally, these transitions are stated as a binary relation between configurations: $\longrightarrow$.

The following transition rule states that if b is *idle* and enabled, then it may change its mode to *working* and consume some inputs. The way that it consumes inputs is one of the choices offered by $\mathit{Consume}(b)$. The conditions above the horizontal line must hold for the transition to be taken; they should be thought of as a hypothesis. That is, if the conditions above horizontal line hold, then the transition below the line may take place [Hen90].

$$
\frac{\begin{array}{c} bm(b) = \mathit{idle},\\ \mathit{Enabled}(b)(fs) = \mathit{true},\\ bm' = [b \mapsto \mathit{working}]bm,\\ (fs', r') \in \mathit{Consume}(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')} \tag{5}
$$

The following transition rule states that if b is *working*, then it may change its mode to *idle* and produce some outputs. The way that it produces outputs is one of the choices

offered by $Produce(b)$.

$$\frac{\begin{array}{c} bm(b) = working, \\ bm' = [b \mapsto idle]bm, \\ (fs', r') \in Produce(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')} \tag{6}$$

To complete the formal semantics of (our extended variant of) DFDs, one would have to describe how to translate the graphical notation into the domains that describe the structure of a DFD (see Table 2). We have not described this translation formally because we did not want to go into the details of the graphical syntax of DFDs [Col91] [Tse91], although we have given an example of one such translation.

Another aspect of the semantics of an (extended) DFD specification is its initial configuration. For simplicity one can make the initial configuration of a DFD have each bubble be idle by default, and to have no values in any read state.

$$bm_{\text{initial}} \stackrel{\text{def}}{=} \lambda b \,.\, idle$$
$$r_{\text{initial}} \stackrel{\text{def}}{=} \lambda b \,.\, \lambda f \,.\, \bot$$

Unfortunately, there is no sensible default initial state for values on flows. One might imagine having a standard *FlowState* mapping, such as $fs_{\text{initial}} = \lambda f \,.\, \langle\rangle$. But if one considers a specification language where users can specify initial values for all flows, such a default makes it difficult to translate that specification language into a DFD. One could try to have each bubbles initialize its outflows upon noting that its inflows are empty, but that fails for sources (as they have no inflows), and also fails when all of a bubble's inflows are initialized by the initialization actions of other bubbles. Hence, we require that the initial *FlowState* mapping be given as part of the definition of a DFD in our model. For our example DFD, an appropriate fs_{initial} would be something like fs_e from Section 3.3.

From a given initial configuration, the above firing rules lead to set of possibly infinite sequences of configurations. Each sequence in the set has the form

$$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \cdots \gamma_n$$

if it is finite or

$$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \cdots \gamma_n \longrightarrow \cdots$$

otherwise. In such a sequence $\gamma_0 = (bm_{\text{initial}}, r_{\text{initial}}, fs_{\text{initial}})$ is the initial configuration, and γ_{i+1} must result from γ_i by the two rules given above.

The set of configuration sequences is the semantics of the given DFD. It is a set because some of the bubbles may be nondeterministic, and because race conditions may result in different firings.

We now briefly consider some alternative semantics that are close to our framework.

The semantics given does not allow any easy way for a bubble to go into an "infinite loop" in working mode. One could imagine letting this be modeled by allowing $Produce(b)(fs, r)$ to be an empty set. However, this is an unsatisfactory way to model such behavior, as then the bubble b cannot give the choice of either going into an infinite loop or doing something else. We leave a satisfactory solution to this problem as future work.

If one prefers not to specify the sources and sinks of a DFD, then one could use a semantics that is a function from specifications (of the sources and sinks) to sets of sequences of configurations. This is easily done, but complicates the formal presentation of the semantics.

Notation for Members $\in$ Name = description	
$f \in F$	$= PowerSet(B) \times FLOWNAMES \times TYPES \times PowerSet(B) \times P$

Table 5: New definition of the F domain.

One might wish to abstract away from the sequences of configurations, in order to focus on the "answer" returned by a DFD. Indeed, this is often done in the semantics of programming languages given in Plotkin's structural style [Plo77] [Hen90]. To do this, one would identify a set of terminal configurations, from which no transitions are possible. However, such an attempt seems of little value for DFDs, because the sources may always be able to fire, as in our example. Even if the sources stop sending outputs into the DFD, the rest of the DFD may be able to continue running. This would be normal in many applications. What can be done, however, is to extract from the sequence of configurations a sequence of values presented to the sinks of a DFD. We leave the investigation of this alternative as future work.

6 Adding Stores

In traditional SA, a *store* is a passive holder of data [DeM78] [Col91]. Multiple bubbles can access a single store for both reading and writing, but the store itself does not transform data. In fact, a store is usually thought of simply as a file, which is a rather low level of abstraction for a specification.

To abstract and formalize the notion of a store, for our theoretical core DFDs we generalize the definition of a flow to subsume stores, following a suggestion by Coleman [Col91, Figure 8.2]. As a flow is also a holder of data, this is reasonable. To capture the idea of multiple bubbles having read or write access to a single store, we allow a flow to have a set of bubble names for its source, and a set of bubble names for its target. Thus, in place of the description of the domain F of flows in Table 2, we now have the description in Table 5.

This change forces changes to several of the auxiliary functions from Section 2.1. We change *Source* and *Target* to *Sources* and *Targets*, respectively. The type of both *Sources* and *Targets* is $F \to PowerSet(B)$. We extend the notions of *input flow* and *output flow* as follows: a flow f is an input flow of bubble b if $b \in Targets(f)$, and f is an output flow of b if $b \in Sources(f)$. The functions *Inputs* and *Outputs* now reflect these new definitions of input and output flows; that is, $Inputs(b)$ is the set of input flows of b, and $Outputs(b)$ is the set of all output flows of b. Using these new definitions, we can now state the requirement that all flows have at least one source and target:

$$(\forall f \in F : Sources(f) \neq \emptyset \wedge Targets(f) \neq \emptyset). \tag{7}$$

The auxiliary functions that describe the structure of a DFD, given in Table 3, are unchanged, except that the *Sources* and *Targets* functions replace the obsolete *Source* and *Target* functions. Surprisingly, the rest of the functions defined in previous sections and the transition rules require no modification to work correctly with this new definition of flows.

With these changes, persistent flows with multiple sources and targets can be used to represent the traditional SA notion of stores. Recall that a bubble may be both the source and target of such a persistent flow, and so can use it as a local store (as was done in our

example). However, allowing multiple sources and targets allows several bubbles to share a store modeled as such a flow. Source bubbles of such a flow are those with write access, and target bubbles are those with read access. As the flow is persistent, any bubble writing to the flow changes the value observed by all the target bubbles. Bubbles reading from the flow do not change the value on it.

This translation of stores into persistent flows also works in the opposite direction. That is, if we had made stores a primitive in the semantics, then we could have translated persistent flows into stores. We chose to say that stores are translated into persistent flows mainly to emphasize the idea of this equivalence. Making persistent flows primary in the model also seems to make for a more uniform semantic model, as flows only go between bubbles, not between stores and bubbles. (However, in a practical DFD specification language, one should certainly include the traditional store notation. In such a language, one might also want to include persistent flows, even though the semantics is equivalent, because stores and persistent flows may have different connotations.)

Stores traditionally represent a collection of data; this is easily modeled by a flow of type set or sequence. Stores containing abstract data types (ADTs) may be modeled by a flow whose type is an ADT. In other words, the type system used in an extended DFD specification language, and not that used in our semantics, is the only thing that would impose a limit on the kind of stores that can be specified.

When consumable flows are allowed to have multiple sources and targets, we find a new semantics with many potential applications. Any of the source bubbles writing to the flow enqueue a new value on it, and any of the target bubbles reading consume the head of the flow. The two-step firing rule and the operational semantics force synchronization: at any transition to a new configuration, only one bubble is active, and this bubble may either read or write, but not both. Hence, no simultaneous read or write is possible. This supports an m-to-n message sending facility that may be useful in specifying systems. For example, one can model an unbounded job queue directly by connecting those bubbles requesting jobs as sources and those servicing jobs as targets to a single consumable flow.

However, this powerful semantics for consumable flows lets one write specifications that may be difficult to implement. An implementer of a specification using consumable flows with more than one source and target must be aware of this semantics to ensure that the proper synchronization is implemented. Whether a specification language should allow such flows is a methodological point, and thus beyond the scope of this paper. We have chosen to make the semantic model more regular, and more expressive, by not requiring flows with multiple sources and targets to be persistent. It is up to a DFD specification language designer to decide if such flows are useful.

7 Bubbles Firing Concurrently with Themselves

To this point in the paper, we have not considered hierarchical DFDs. However, when one bubble is refined into a sub-DFD containing multiple bubbles, the sub-DFD often produces results that would not have been possible in the original DFD. For example, consider the refinement shown in Figure 5. As we prefer not to discuss specific syntax for firing rules, we describe what the bubbles in this figure do informally. Informally, bubble `H` is a top-level specification of the function

$$from = \textbf{if } to \leq 2 \textbf{ then } to^2 \textbf{ else } to^3 \textbf{ fi}$$

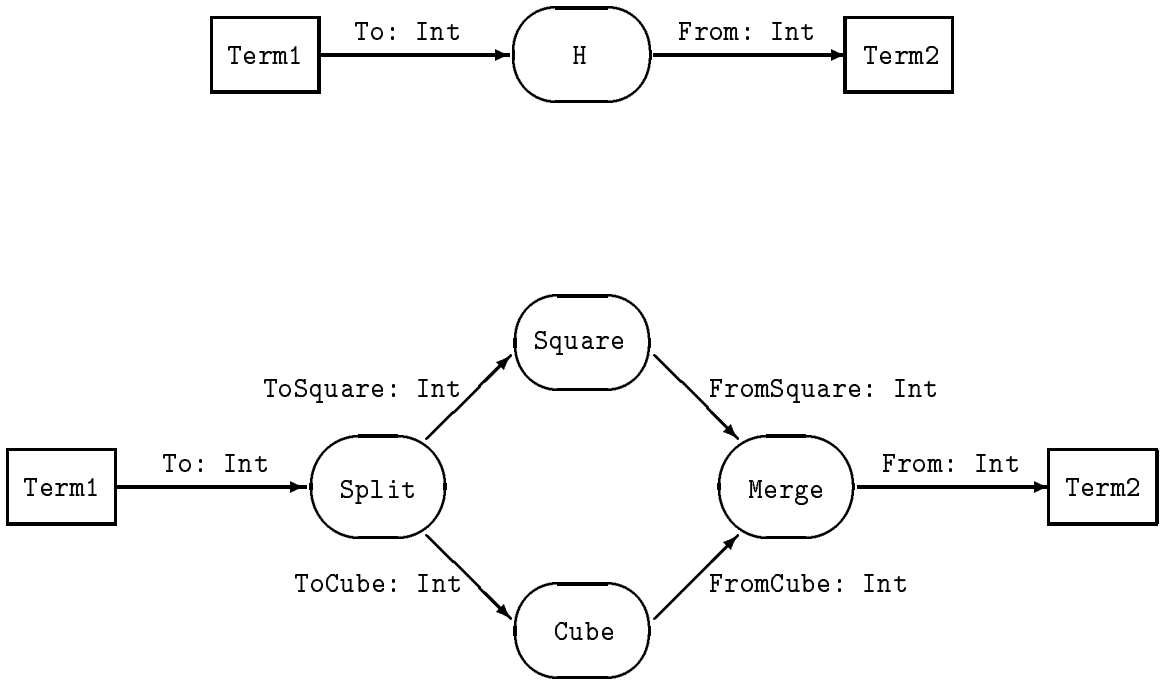

Figure 5: An example of refinement.

where *to* is the value read from the flow named `To` and to^2 or to^3, as appropriate, is written to the flow `From`. The rest of the figure is intended as a refinement of bubble `H`. Bubble `Split` sends inputs less than or equal to 2 to bubble `Square`, and other inputs to bubble `Cube`. `Square` and `Cube` each read their inflow, compute the appropriate value, and write it to their outflow. Bubble `Merge` reads from either of its inflows, and places the read value on the flow named `From`.

Intuitively, the refinement should behave exactly like the original bubble. However, if the flow named `To` contains integers 2 and 3, then the original bubble will always place 4 on the flow named `From` before placing 27 there, whereas either order is possible with the refinement. To see this, consider the case where `Split` reads 2 from `To` and writes 2 to `ToSquare`, and then reads 3 and writes it to `ToCube`. Next, `Cube` reads 3 and writes 27 to `FromCube`, followed by `Merge` reading this value from `FromCube` and writing it to `From`. Then `Square` reads 2 from `ToSquare` and writes 4 to `FromSquare`. Finally, `Merge` reads 4 from `FromSquare` and writes it to `From`. While this is clearly a contrived example, this situation can arise any time one bubble is refined by multiple bubbles.

If the bubble `H` could somehow hold the first value it consumes while consuming and producing output with the second, then the two DFDs would behave identically [Lyl92]. This is our notion of a bubble firing concurrently with itself — a bubble can consume from its input flows multiple times without producing any output, and when the bubble does produce output, the output can be produced from the input provided by any of the consumptions, not just the first. One might want to specify `H` with such a semantics precisely to allow it to be refined as in Figure 5.

We formalize this notion of a bubble firing concurrently with itself by changing the type

The following must hold for all $b \in B$, $fs \in FlowState$, and $r \in Read$.

[c-range:] If $Enabled(b)(fs) = true$, then $Consume(b)(fs, r) \neq \emptyset$.

[c-local:] If $(fs', r') \in Consume(b)(fs, r)$, then

- for all $b' \neq b$, $r'(b') = r(b')$ and
- for all $f \in F$:
  - if $f \notin Inputs(b)$ then $fs'(f) = fs(f)$, and
  - if $f \in Inputs(b)$ then $r'(b) = \{g\} \uplus r(b)$ and fs' and g are such that
    - if $Consumable(f)$, then either:
      - $fs'(f) = fs(f)$ and $g(f) = \perp$, or
      - $IsEmpty(fs(f)) = false$ and $fs'(f) = Rest(fs(f))$ and $g(f) = Head(fs(f))$,
    - if $\neg Consumable(f)$, then $fs'(f) = fs(f)$ and either:
      - $g(f) = \perp$, or
      - $IsEmpty(fs(f)) = false$ and $g(f) = Head(fs(f))$.

Figure 6: New restrictions on *Consume* allowing for bubbles to fire concurrently with themselves.

of read mappings to be:

$$Read = B \rightarrow MultiSet(WhatRead). \tag{8}$$

Recall that $WhatRead = (F \rightarrow OBJECTS_{\perp})$. That is, a read mapping r is a function from bubble names to multisets, such that for each bubble name b, $r(b)$ represents what b has consumed but not used to produce output. If $g \in r(b)$, then for each flow f, $g(f)$ represents an object that b has read from the flow f, or is $\perp$ if b did not read from f during that consumption. Note that if f_1 and f_2 are flows, then $g(f_1)$ and $g(f_2)$ were read at the same time when b read from its inputs. Formerly $r(b)$ was a single function from flows to objects; now the semantics tracks several such functions.

This change in the type *Read* forces changes in the restrictions on the *Enabled*, *Consume*, and *Produce* mappings. As a bubble need not produce between consumings, we need to apply the *Enabled* and *Consume* mappings to *working* bubbles, as well was to *idle* ones. Otherwise, *Enabled* need not be changed.

The changes required for *Consume* are more substantial. In dealing with the multisets introduced by the new definition of *Read*, we use the symbol "$\uplus$" for multiset union, and "$-$" for multiset difference. Figure 6 reproduces Figure 3 with the appropriate modifications.

The function *In*, used in examples of *Consume* mappings, changes in the corresponding way.

$In : (F \times B) \rightarrow ((FlowState \times Read) \rightarrow (FlowState \times Read))$
$In(f, b)(fs, r) =$
  let $g_b = \lambda f \,.\, Head(fs(f))$

in (**if** $Consumable(f)$ **then** $[f \mapsto Rest(fs(f))]fs$ **else** fs **fi**,
$[b \mapsto (\{g_b\} \uplus r(b))]r$)

Changes are also required in the restrictions on the *Produce* mapping. The appropriately modified version of Figure 4 is given as Figure 7. The only major change is in the **p-depend** restriction. The new version says that what a bubble b outputs depends only on which element of the multiset $r(b)$ is used when b produces. Other elements of $r(b)$ do not affect the output.

Similarly, the function *Out*, used in examples of *Produce* mappings, must also be modified to account for the redefinition of the *Read* domain. As the range of the *Read* domain is now a multiset, the intuitive idea of removing read information that has been used to produce output corresponds to multiset difference. However, as the range of *Read* is a *multi*set, for some bubble b, function g of type *WhatRead* can appear in $r(b)$ multiple times. Thus, if this new *Out* function is composed with itself, a term of the form $[b \mapsto r(b) - \{g\}]r$ in the *Out* function would result in two occurrences of g in the multiset being removed, rather than just one occurrence as is probably intended. Hence, we handle updating the *Read* mapping explicitly in the *Produce* mapping, as shown in the clause for bubble `H` in the example below.

$Out : (OBJECTS \times F \times B) \to ((FlowState \times Read) \to (FlowState \times Read))$
$Out(o, f, b)(fs, r) =$
(**if** $Consumable(f)$ **then** $[f \mapsto Enq(fs(f), o)]fs$ **else** $[f \mapsto Enq(\langle\rangle, o)]fs$ **fi**, r)

Thus, a possible *Produce* mapping for the first DFD in Figure 5, using the flow abbreviation style of Table 1, is:

$Produce_e(\texttt{Term1}) = \lambda(fs, r)\,.$
$\{Out(i, f_{\texttt{To}}, \texttt{Term1})(fs, r) \mid i \in TypeMeaning(\texttt{Int})\}$
$Produce_e(\texttt{H}) = \lambda(fs, r)\,.$
{**let** $to = g(f_{\texttt{To}})$ **in**
$Out($**if** $to \leq 2$ **then** to^2 **else** to^3 **fi**$, f_{\texttt{From}}, \texttt{H})(fs, [\texttt{H} \mapsto r(\texttt{H}) - \{g\}]r)$
$\mid g \in r(\texttt{H})\}$
$Produce_e(\texttt{Term2}) = \lambda(fs, r)\,.\,\{(fs, [\texttt{Term2} \mapsto \{\}]r)\}$

With these changes in the *Enabled*, *Consume*, and *Produce* mappings, the new transition rules required are straightforward. A *working* bubble may now consume, so the first transition rule of Section 5 is no longer restricted to operating on *idle* bubbles. Note that the update to the *BubbleMode* of b has no effect if b was already a *working* bubble.

$$\frac{\begin{array}{c} Enabled(b)(fs) = true, \\ bm' = [b \mapsto working]bm, \\ (fs', r') \in Consume(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')} \qquad (11)$$

There are now two possibilities when a bubble produces output. If the bubble has consumed only once more than it has produced output (i.e., $|r(b)| = 1$), then the bubble becomes *idle*. Otherwise, it remains *working*. Note that if the mode of a bubble is *working*, then $|r(b)| \geq 1$.

The following must hold for all $b \in B$, $fs \in FlowState$, and $r \in Read$.

[p-range:] $Produce(b)(fs, r) \neq \emptyset$.

[p-local:] If $(fs', r') \in Produce(b)(fs, r)$, then

- for all $b' \neq b$, $r'(b) = r(b)$,
- for some $g \in r(b)$, $r'(b) = r(b) - \{g\}$, and
- for all $f \in F$:
  - if $f \notin Outputs(b)$ then $fs'(f) = fs(f)$, and
  - if $f \in Outputs(b)$, then for some $o \in TypeMeaning(TypeOf(f))$:
    - if $Consumable(f)$, then either:
      - $fs'(f) = fs(f)$, or
      - $fs'(f) = Enq(fs(f), o)$,
    - if $\neg Consumable(f)$, then either:
      - $fs'(f) = fs(f)$, or
      - $fs'(f) = Enq(\langle\rangle, o)$.

[p-depend:] For all $fs' \in FlowState$, for all $r' \in Read$, if there is some $g \in r'(b)$ such that $g \in r(b)$, then there is a one-to-one correspondence between the following two sets

$$\{(fs_0, r_0) | r_0 = [b \mapsto (r(b) - \{g\})]r \text{ and } (fs_0, r_0) \in Produce(b)(fs, r)\} \tag{9}$$

and

$$\{(fs_1, r_1) | r_1 = [b \mapsto (r'(b) - \{g\})]r' \text{ and } (fs_1, r_1) \in Produce(b)(fs', r')\} \tag{10}$$

such that: for all (fs_0, r_0) in the first set and for the corresponding (fs_1, r_1) in the second, for all $f \in Outputs(b)$, and for all $o \in TypeMeaning(TypeOf(f))$,

- $fs_0(f) \neq fs(f)$ if and only if $fs_1(f) \neq fs'(f)$, and
- either $fs_0(f) = Enq(fs(f), o)$ or $fs_0(f) = Enq(\langle\rangle, o)$ if and only if either $fs_1(f) = Enq(fs'(f), o)$ or $fs_1(f) = Enq(\langle\rangle, o)$.

Figure 7: New restrictions on *Produce* allowing for bubbles to fire concurrently with themselves.

$$\frac{\begin{array}{c} bm(b) = \mathit{working}, \\ bm' = (\textbf{if } |r(b)| = 1 \textbf{ then } [b \mapsto \mathit{idle}]bm \textbf{ else } bm \textbf{ fi}), \\ (fs', r') \in \mathit{Produce}(b)(fs, r) \end{array}}{(bm, r, fs) \longrightarrow (bm', r', fs')} \tag{12}$$

Finally, to support the change in the type *Read*, the initial read mapping must return an empty multiset for each bubble, rather than the function $\lambda f \,.\, \bot$.

$$r_{\text{initial}} \stackrel{\text{def}}{=} \lambda b \,.\, \lambda f \,.\, \{\}$$

8 Discussion

In this section we discuss related work and offer some conclusions.

8.1 Related Work

We have already discussed some related work that describes ways to derive formal specifications from the products of SA, and which augments SA ERDs and DFDs with formal specifications. In this paper we have done neither of these things, although they are motivation for our work. Instead, we have defined a formal semantics for a "theoretical core variant of DFDs" that includes a way to model the dynamic behavior of such DFDs.

In what follows we discuss work that defined various other notions of DFDs, including the original notions. We try to point out the differences from our notion of DFDs; as we mentioned in the introduction, such differences should not be seen as a defect in our work, because in each case the features not present in our models can be translated into our models. We also point out places where related work does not give a precise semantics, especially for dynamic behavior. In this part of the paper, "DFD" will no longer mean our theoretical core variant of DFD, but the traditional notion.

To summarize, the main advantage of our work is greater rigor, greater expressive power, and the ability to model dynamic behavior.

8.1.1 De Marco and Yourdon

DFDs used in traditional structured analysis [DeM78] [Col91, page 15] have a very informal flavor and some features that are not directly present in our models. For example, De Marco has graphical notations on flows for "conjunction" (*) and "disjunction" ($\oplus$) that are not a part of our formal model of the syntax of DFDs. Similarly, Yourdon makes graphical distinctions between "data" and "control" flows [You89] [Col91, pages 27–28]. However, our formal model can encode this kind of information in a general way. Our model of P-specs allows one to say that a bubble consumes or produces on two flows at once, or one only; the advantage of our formal model is that it can also express any other computable way that bubbles could consume or produce on flows. Our formal model can express control flows in a variety of ways; for example, by using a persistent flow that transmits boolean values, or by using a consumable flow that transmits values of a one-element type (a signal).

De Marco's DFDs feature converging and diverging flows [DeM78]. However, their semantics is ambiguous. A converging flow may mean either that:

- there are several flows which all have their sinks in the same bubble, or

- several "elementary packets of data" are to be joined "to form a complex packet" (such as a tuple) [Col91, page 16].

In our model one can express the first by having several separate flows, with a flow having several sources as in Section 6. In our model, one can express the second meaning for converging flows by introducing a bubble to accept each of the "converging" flows and specifying how it combines them. This avoids any possible ambiguity. De Marco's diverging flows are ambiguous in the same way. In our model one can express such a diverging flow either as: several flows (one to each destination of a De Marco style diverging flow), with a flow having several targets as in Section 6, or with a flow into a bubble that splits the data into parts, and sends each part on a different output flow. Again our model does not suffer from the ambiguity inherent in De Marco's DFDs. Ward, in [War86] gives notation to disambiguate these two senses of converging and diverging flows, but leaves the process by which tupling and splitting of data is achieved implicit. Similarly, our model can represent "dialogue flows" by two separate flows.

A feature of De Marco's DFDs that we do not model directly is the description of a system as a hierarchy of DFDs. A hierarchical DFD can be considered as a graphical convenience, by expanding it out into a single level DFD, which can then be given a semantics using our model. Nevertheless, it would be useful to have an operational semantics of DFDs that took the hierarchy into account (as in the semantics of hierarchical colored petri nets [HJS90] [CJ91]).

8.1.2 Ward

Ward [War86] distinguishes discrete from continuous data flows; that is between discrete and analog data, which he defines as "a set of values defined continuously over a time interval" [War86, page 199]. Hatley and Pirbhai have a similar notion of continuity [HP87] [Col91, Section 5.3.2]. Since such data can only be modeled in a computer by discrete data, there seems to be no good reason to model this distinction. Instead, we have adopted Ward's semantics for continuous flows as shared variables [War86, page 203] as a feature of flows in our model. This feature of flows is orthogonal to the type of data on the flow. This is the distinction between consumable and persistent flows [Col91, Section 5.3.4].

The traditional view of stores is that they represent files [War86, page 199]. We model stores as persistent flows (shared variables), but as discussed in Section 6, the notion of a store containing a single value and a shared variable are equivalent. Our semantics is more general, in that the type of such a shared variable might be a "file type" but it could also be more specific, such as an integer.

Ward also includes in his extension of DFDs something called a "buffer", which is highly ambiguous. He says that a "buffer is an abstraction on a stack or a queue" [War86, page 200]. Stacks and (LIFO) queues certainly have different behavior.

Ward gives a semantics of DFDs based "loosely on the execution of a Petri net" [War86, pages 203–205]. However, his semantics are somewhat informal and ambiguous. For example, he does not clear up the ambiguity in the potential behavior of buffers. Ward does not discuss an initial marking of the DFD with tokens, and his semantics does not deal with the actual values of data on flows. Hence his semantics can only be approximate (a kind of abstract interpretation of the DFD), because the transitions between markings cannot depend on the values of data on the flows or the exact functions computed by bubbles. Our semantics can be regarded as a cleaned-up version of this idea.

8.1.3 Tse and Pong

Tse and Pong recognize that: "Transitions and places of Petri nets correspond, respectively, to processes and data flows of DFDs" [TP89, page 1]. They also give an algebraic model of DFDs. However, their semantics using Petri nets also ignores the values on the flows and so cannot describe the full behavior of a DFD.

Tse's work [Tse91] only deals with the syntax of DFDs, not their semantics.

8.1.4 Colored Petri Nets

One can imagine giving a fuller account of the dynamic behavior of DFDs by extending Ward and Tse and Pong's approach with colored Petri nets [Jen91] [HJS90]. (The different "colors" on tokens can stand for different values being passed in a DFD.) However, because the firing rules of Petri nets are atomic, each bubble in a DFD would have to be either modeled by a complex Petri net, or the semantics of a bubble's firing would be atomic. This is because in a Petri net, a transition reads from its input places and writes its output places in an atomic step, and so using Tse and Pong's idea would imply that a bubble in a DFD would reads its input flows and write its output flows in an atomic step. However, as described in Section 3.1, such a semantics is inadequate for capturing the dynamic behavior of a DFD.

8.1.5 Coleman

The static semantics of SA specifications (including notational issues) are treated in a recent dissertation by Coleman [Col91]. As noted above, we have drawn on this dissertation for its comprehensive discussion of the literature on DFDs and SA, for our model of flows as having both a name and a type, for the distinction between consumable and persistent flows and for the semantics of consumable and persistent flows. Coleman describes a notation for P-specs based on first-order logic [Col91, Chapter 7], which inspired parts of our model; his notation could be translated into our model.

Coleman also describes how one could give an operational semantics of SA specifications [Col91, Section 9.2] using what amounts to colored Petri nets (values are used instead of Ward's tokens) and first-order logical assertions for the transition firing rules. As described above in the section on colored Petri nets, however, such a semantics is inadequate for modeling the dynamic behavior of DFDs. Nevertheless, our model is derived from his initial work.

Coleman also gives several different possible models of stores, one of which [Col91, Figure 8.2] is the one we adopt. However, he ignores stores in his sketch of the operational semantics of DFDs. Coleman does not treat a bubble firing concurrently with itself.

8.1.6 France

In the two works [Fra92] [Fra93] that we discuss below, France describes variants of DFDs, specification notations, and their semantics. Although his semantics are not general enough for our purposes, this is understandable, because his papers did not seek to give a general foundation for extended DFD specification languages; instead, his papers seek to give a semantics for the particular DFD variants and specification notations. As with the other work cited here, he is not as concerned with the dynamic behavior of DFDs as we are. In summary, the major difference is that France presents specific specification notations

and their semantics, while we have explored what kinds of semantics for extended DFD specification are sensible, with a special focus on modeling the dynamic behavior of DFDs.

Semantically Extended DFDs In [Fra92] France works with DFDs that have several additional features, and he also gives a formal semantics. France's "queued flows" are what we call consumable flows, and his "variables" are our persistent flows. France distinguishes two kinds of bubbles (data transforms and state transforms) and two kinds of flows (data and control). Our model is arguably simpler in that both of his types of bubbles and flows can be translated into ours.

France's operational semantics of extended DFDs use a framework similar to ours: algebraic state transition systems (ASTSs) [AR91] [Ast91]. The main difference in the formal framework is that France's semantics use transition systems to model all parts of a DFD, not just the firing rules. This gives the semantics a nicely compositional flavor, and allows an easy treatment of hierarchical DFDs [Fra92, page 333]. France gives an explicit notation for P-specs, which could be given an alternative semantics using our model. All of his additional graphical conventions for DFDs can be translated into our model in ways similar to those discussed above for other related work. France's DFDs have syntactic and semantic restrictions that seem to be intended to enforce good methodology, but which also make them less general than ours. For example, each bubble must have an output flow (page 330) and nested loops are not allowed inside infinite loops in his P-spec statement language (page 337).

France treats stores the same as he treats other parts of a DFD, as ASTSs [Fra92, pages 336–337]; hence his stores need more description than necessary in our model, where stores are modeled as persistent flows. However, France's stores can be more powerful than this, as he points out that one can "specify that a subset of write actions have priority over a subset of read actions when they occur in parallel" (p. 337). Nevertheless, by translating such an active store into a bubble with a persistent flow to and from itself, such a store can be translated into our model.

France's notion of convergence and divergence is apparently the same as Ward and Mellor's (see France's Figure 2, p. 330). France calls these "binders" and "splitters". Since he does not treat them in his formal semantics, his notions appear to suffer the same problems (no precise description of how the splitting or aggregation is done).

France does not allow bubbles to fire concurrently with themselves because each bubble has only one copy of its variables [Fra92, pages 344–345]. Firing in a DFD is apparently an atomic step, since "outputs are solely dependent on current inputs" [Fra92, page 342]; so France's model includes explicit ways to make bubbles concurrent. However, there is no way to make a bubble concurrent with itself, and so no way to specify a bubble that may be refined into as system of bubbles that may process inputs in slightly different orders, which we permit by allowing a bubble to fire concurrently with itself [Lyl92].

A Predicative Basis for SA Specification Tools In [Fra93] France gives a different formalization of the semantics of DFDs, this one geared towards composition and decomposition of DFDs. In this work France uses a variant of DFDs he calls PDFDs, along with ERDs, and data dictionaries with ADT specifications. His PDFDs include terminators (external entities), as do ours.

In [Fra93], the behavior of a bubble is specified with an input/output predicate, which can be represented in our model as the functions that tell what a bubble produces and

consumes. However, his language is not able to express conditions on when a bubble is enabled.

The major difference between his semantics for a PDFD and our model is that his semantics regards a PDFD as an atomic data transformation. That is, his semantics says what outputs (and global state changes) a bubble or a PDFD can produce for a single input. Such data transformations are atomic, because they cannot model time delays and race conditions, as discussed in Section 3.1 above. While for some purposes this might be adequate to characterize the behavior of a single bubble (and for his example they seem adequate), it is inadequate as a general model for the dynamic behavior of entire DFDs, because it does not allow for concurrency within a DFD. For the same reason, France's composition operators are also inadequate as a basis for studying the dynamic behavior of DFDs: they operate on single inputs and generate an atomic data transformation. For example, the parallel composition operator does not allow one PDFD to run twice while the other waits.

Because of this atomic view of the PDFD semantics, France's notion of the correctness of a decomposition is too restrictive for a semantics of DFDs that is concerned with dynamic behavior, as described in Section 7 above.

8.2 Conclusions

In this paper we have described a formal foundation for the semantics of extended DFD specifications by giving a structural operational semantics of a theoretical core variant of DFDs. Our formalism should not be considered a proposal for notation that anyone would use in practice, as indeed we have tried to avoid describing P-spec and DFD notations. Instead we have described semantics that can adequately capture the meanings of various extended DFD specification languages. In particular we have focused on capturing the dynamic behavior of DFDs.

The main problem with practical application of this work is to give a syntax for P-specs and to give its formal semantics using these ideas.

However, our work does have implications for practitioners and specification language designers. Our work on a precise formal semantics for DFDs provides the following practical insights.

- One can use DFDs to specify the dynamic behavior of a system. We showed how to specify this behavior by adding the following information to DFDs:
  - when each bubble is enabled
  - what the bubble reads when it is enabled
  - what the bubble produces when it finishes its work, and
  - the initial state of the system.
- In addition to flow names, one should put the types of data as labels on the flows, and specify these types as ADTs in the data dictionary. Having both the name and type on a flow prevents ambiguity and conveys more information. Allowing the types to be arbitrary ADTs permits modern software engineering practices to be used and helps keep the level of abstraction high.
- There is no fundamental difference between a control flow (or control bubble) and a data flow (or data bubble). Thus one can use data flows to achieve some measure of

control, and designs without explicit control flows are not necessarily less worthy than those with explicit control flows.

- One can think of stores abstractly as shared variables, not just as files. Allowing a store to have an arbitrary abstract data type gives the design more flexibility and keeps implementation details out. (Of course, one useful ADT is a file, so no expressive power is lost.)

- One should think of a bubble as a collection of cooperating processes — not as a procedure. This allows DFDs to be useful as high-level specifications for concurrent and distributed systems, and it keeps the DFD from becoming embroiled in low-level procedural details. The ADTs used by the DFD can (and will often) be implemented with procedures. Thinking of a bubble as a collection of cooperating processes recognizes that each bubble may be refined into another DFD. This is supported by our semantics which allows a bubble to fire concurrently with itself.

In some discussions we have heard it said that it is counter-productive to try to formalize DFDs: aren't DFDs supposed to be an informal notation that is intended to be understood by customers? Wouldn't a formal notation be harder for a customer to understand? We believe that customers think they understand the DFD notation, but often understand something different than what was intended. We know that as an ambiguous, informal notation, DFDs are open to differing interpretations, sometimes with disastrous results. However, we do not want to stop people from using DFDs informally. What we want is to have the possibility to use some extension of DFDs in a precise, formal way. This precise use of DFDs would certainly come after the imprecise, informal use, but one should not have to completely change notations in order to formally describe systems. Instead, we look forward to an integration of informal and formal specifications spanning a wide range of needs [FLP94] [FLP95]. The advantage of formal and precise DFD specifications would be that the work done in requirements analysis would not have to be thrown away when more precision is required.

Acknowledgements

Thanks to Juergen Symanzik for helpful suggestions and corrections, especially about strictness in the semantics. Thanks to two anonymous referees (of an earlier version of this paper) for corrections and helpful comments. Thanks to Joe Reynolds for help with taking notes and initial typing. Thanks to Joe, and the rest of the seminar on Structured Analysis in Spring '91 (Joe Reynolds, John Rose, Becky Wemhoff) and Spring '92 (Joe Reynolds, Bashar Jano, Soma Chaudhuri) for help in developing the formal models.

References

[AR91] E. Astesiano and G. Reggio. SMoLCS Driven Concurrent Calculi. In Hartmut Ehrig et al., editors, *TAPSOFT'87, Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, Volume 1*, volume 245 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1991.

[Ast91] Edigio Astesiano. Inductive and Operational Semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, New York, N.Y., 1991.

[BB93] Jorgen P. Bansler and Keld Bodker. A Reappraisal of Structured Analysis: Design in an Organizational Context. *ACM Transactions on Office Information Systems*, 11(2):165–193, 1993.

[BJ82] Dines Bjorner and Cliff B. Jones. *Formal Specification and Software Development*. International Series in Computer Science. Prentice-Hall, Inc., London, 1982.

[Chu41] A. Church. *The Calculi of Lambda Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, N.J., 1941. Reprinted by Klaus Reprint Corp., New York in 1965.

[CJ91] Soren Christensen and Leif Obel Jepsen. Modelling and Simulation of a Network Management System using Hierarchical Coloured Petri Nets (extended version). Technical Report DAIMI PB 349, Computer Science Department, Aarhus University, Apr 1991.

[Col91] David L. Coleman. *Formalized structured analysis specifications*. PhD thesis, Iowa State University, Ames, Iowa, 50011, 1991.

[DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon , Inc., Englewood Cliffs, New Jersey, 1978.

[FKV91] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and Foraml Requirements Specification Languages: Bridging the Gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

[FLP94] Robert B. France and Maria M. Larrondo-Petrie. From Structured Analysis to Formal Specifications: State of the Theory. In *Proceedings of the ACM Computer Science Conference, Phoenix, AZ*, pages 249–256. ACM, Mar 1994.

[FLP95] Robert B. France and Maria M. Larrondo-Petrie. A Two-Dimensional View of Integrated Formal and Informal Specifications Techniques. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland*, volume 967 of *Lecture Notes in Computer Science*, pages 434–448. Springer-Verlag, September 1995.

[Fra92] Robert B. France. Semantically Extended Data Flow Diagrams: A Formal Specification Tool. *IEEE Transactions on Software Engineering*, 18(4):329–346, April 1992.

[Fra93] R. B. France. A predicative basis for structured analysis specification tools. *Information and Software Technology*, 35(2):67–77, February 1993.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[GS78] C. Gane and E. Sarson. *Structured Systems Analysis: tools and techniques.* Prentice-Hall, 1978.

[Hay93] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics.* John Wiley and Sons, New York, N.Y., 1990.

[HJS90] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in Coloured Petri Nets. In G. Rosenberg, editor, *Advances in Petra nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1990.

[HP87] D. J. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, N.Y., 1987.

[Jen91] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Analysis and Design. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. Also a technical report from the CS Dept, Aarhus University, DAIMI PB-338, Nov. 1990.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[LvKP+91] Peter Gorm Larsen, Jan van Katwijk, Nico Plat, Kees Pronk, and Hans Toetenel. SVDM: An integrated combination of SA and VDM. In *Methods Integration Conference*. Springer-Verlag, September 1991.

[Lyl92] Kari Ann Lyle. Refinement in Data Flow Diagrams. Master's thesis, Iowa State University, Ames, Iowa 50011, July 1992.

[Pet77] J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):221–252, September 1977.

[Plo77] G. D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[SA91] L. Semmens and P. Allen. Using Yourdon and Z: An Approach to Formal Specification. In *Z User Workshop*. Springer-Verlag, 1991.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[Spi89] J. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, January 1989.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[TP89] T. H. Tse and L. Pong. Towards a Formal Foundation for Demarco Data Flow Diagrams. *The Computer Journal*, 32(1):1–12, February 1989.

[Tse91] T. H. Tse. *A Unifying Framework for Structured Analysis and Design Models*, volume 11 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univerity Press, New York, N.Y., 1991.

[War86] Paul T. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[WBL93] Tim Wahls, Albert L. Baker, and Gary T. Leavens. An Executable Semantics for a Formalized Data Flow Diagram Specification Language. Technical Report 93-27, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, November 1993. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*, volume 1: Introduction and Tools. Yourdon, Inc., Englewood Cliffs, New Jersey, 1985.

[WZ91] Jeannette M. Wing and Amy Moormann Zaremski. Unintrusive Ways to Integrate Formal Specifications in Practice. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 545–569. Springer-Verlag, New York, N.Y., October 1991.

[You89] Edward Yourdon. *Modern Structured Analysis*. Yourdon Press computing series. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.